Technical University of Vienna
Information Systems Institute
Distributed Systems Group

End-to-End Versioning Support for Web Services

(This technical report has been submitted to the SCC08 conference.)

Philipp Leitner, Anton Michlmayr,
Florian Rosenberg, Schahram Dustdar
leitner@infosys.tuwien.ac.at
michlmayr@infosys.tuwien.ac.at
florian@infosys.tuwien.ac.at
dustdar@infosys.tuwien.ac.at

TUV-1841-2008-1 March 11, 2008

Software services are, just like any other software system, subject to permanent change. We argue that these changes should generally be transparent to service consumers. However, currently consumers are often tied to a given version of a service and have no means of easily upgrading to a newer version. In this paper we propose a WSDL-driven classification of Web service change types and discuss a versioning mechanism for service-oriented systems that considers revision management on registry- and client-side. We use the concepts of service version graphs and selection strategies to provide transparent end-to-end versioning support, and show how this approach is implemented in our service-oriented computing runtime VRESCo. Furthermore, we illustrate the advantages of our approach in comparison to the current state of the art using a realistic case study.

Keywords: SOA, Web Services, VRESCo, Versioning, Evolution

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
URL: http://www.infosys.tuwien.ac.at/

End-to-End Versioning Support for Web Services

Philipp Leitner, Anton Michlmayr, Florian Rosenberg, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
lastname@infosys.tuwien.ac.at

Abstract

Software services are, just like any other software system, subject to permanent change. We argue that these changes should generally be transparent to service consumers. However, currently consumers are often tied to a given version of a service and have no means of easily upgrading to a newer version. In this paper we propose a WSDL-driven classification of Web service change types and discuss a versioning mechanism for service-oriented systems that considers revision management on registry- and client-side. We use the concepts of service version graphs and selection strategies to provide transparent end-to-end versioning support, and show how this approach is implemented in our service-oriented computing runtime VRESCo. Furthermore, we illustrate the advantages of our approach in comparison to the current state of the art using a realistic case study.

1. Introduction

Software systems in the real world are subject to permanent change – vendors constantly add new functionality or change the requirements of existing applications, and strive to increase quality aspects such as reliability or security. This software adaptation process is usually referred to as "software evolution", and is subject to a vital research community [1].

With the advent of service-oriented architectures (SOA) [12] one could believe that evolution of services is no longer an important issue since services have a dedicated service contract and evolution aspects can be hidden from the service requesters. However, all too often new requirements and environmental or technological changes still lead to modifications of the contract. Clearly, such modifications revive the problem of maintaining multiple versions of a service. Additionally, new facets are added to the versioning problem in SOA environments. On the one hand, service providers often want to provide several versions in parallel, offering specific variants to some customers or older service versions for legacy applications. On the other hand, some service requesters may want to access different service versions in a uniform manner or even switch between them at runtime, while others do not want to explicitly deal with different service versions. Therefore, the main issue besides managing multiple versions of a service in the registry is to provide a certain degree of version transparency towards clients. The latter should be able to dynamically invoke different versions at runtime without having to modify their code base.

In the current Web services stack service contract changes require all service requesters to re-engineer their applications to ensure that they conform to the new contract. More precisely, current service registries such as UDDI [11] and the ebXML registry [10] do not provide support for all facets of service evolution. These registries use a flat service model that does not consider service variants, or multiple versions of the same base service. Furthermore, the issues of binding and mediating between different service versions are not addressed by pure registry technologies at all, and are left entirely to the service requesters. In contrast to these standards, we argue that managing Web service evolution in an end-to-end fashion should be a core feature of any real-world SOA solution.

The contributions of this paper are threefold: firstly, we present a classification of various service change types; secondly, we introduce a general versioning approach to manage evolutionary changes in Web services within Web service registries, and thirdly, we propose a client-side approach using proxies that enables transparent binding and mediation between different versions of a service. We have implemented our service versioning approach within our VRESCo SOA runtime environment [8] and evaluate our implementation based on a realistic, yet simple case study from the telecommunications domain.

The remainder of this paper is structured as follows: Section 2 discusses a classification of possible evolution-

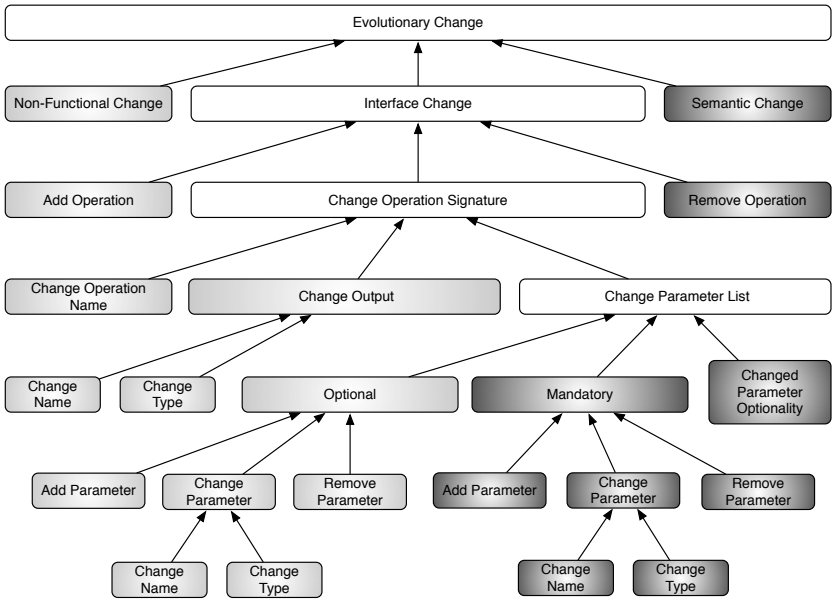

Figure 1. Version change classification scheme

ary changes of Web services and how these changes can be managed, Section 3 introduces the VRESCo runtime environment and explains how it supports end-to-end versioning concepts, Section 4 discusses the advantages of our approach compared to a solution based on state-of-the-art technology. Section 5 presents some related work in the field, while Section 6 concludes the paper and gives an outlook to future work.

2. Service Versioning Concepts

Web service evolution is the process of Web services being advanced, adapted, and adjusted over time. In this process, not only the newest revision[1] will be available. Service providers might often want to keep older versions of their services online for some time to keep compliance with existing clients. In this section we present a WSDL-driven [17] taxonomy of possible changes of Web services, discuss service version graphs as a means of representing Web service evolution and explain how clients may operate on these graphs.

2.1. Classification of Version Changes

A classification scheme for differences between Web services has been proposed by Ponnekanti et al. [13]. They specify four types of incompatibilities: structural, value, encoding, and semantics. We base our taxonomy on their approach, but refined it to more closely relate to Web service versioning instead of differences between independent services. Another distinction is that we do not compare service versions on SOAP [19] message level, but rather based on their WSDL descriptions.

Before describing the types of changes, we have to define the concept of *functional interfaces* of services. The functional interface contains all operations that are defined in the services' WSDL description. Every operation in the functional interface consists of an operation name, a list of parameters and an output. Both parameters and output consist of a name and a type. Parameters may either be optional or mandatory. All other properties of a service (e.g., the endpoint address, the WSDL encoding style, the policy that has to be adhered to) are considered to be part of the *non-functional interface* of the service.

Our classification scheme is shown in Figure 1. We distinguish three top-level types of changes: (1) *Non-Functional Changes* are all changes in the non-functional interface of the Web service as defined above, (2) *Interface Changes* represent all changes in the functional interface, and (3) *Semantic Changes* cover all changes that are not contained in the WSDL description of the service, such as a changed understanding (but not changed structure) of operations, parameters or return values.

Interface Changes are relatively structured, so we give a further breakdown of this change type in Figure 1. Non-functional changes cover a wide range of aspect such as QoS attributes (as described in [14]) and service policies. It is important to note that not all non-functional changes

[1] We use the terms version and revision interchangeably in this paper.

require the definition of a new service version. For example, QoS attributes typically represent dynamic values which change frequently and continuously, therefore, they are not covered by service versioning. A complementary approach that addresses service adaptation based on QoS is described in [9].

In this paper, we do not cover semantic changes since they would require a semantically annotated WSDL (as proposed, for example, by SAWSDL [18]) which is currently not available in real-world applications.

The different shades of gray in Figure 1 represent the transparency of the change categories, and will be further explained in Section 3.

2.2. Service Version Graphs

In order to manage Web service evolution, service registries need to store not only the service revisions itself, but also how they relate to each other. We use the notion of *service version graphs* to represent these dependencies. For every service in the registry there is exactly one service version graph. These graphs are *directed*, with nodes representing concrete revisions of the service, and edges representing *predecessor-successor* relationships. The semantics of these relationships is that revision A is a predecessor of revision B, if B is the result of changes in A. All revisions in a service version graph refer to the same base service, but are on different maturity levels and represent different stages in the base service's lifecycle.

Service version graphs may contain *branches* and *merges*. Branches represent situations where two or more variants of a base service evolve in parallel, for instance if a special alternative service version with specific behavior has to be created for a subset of users. Merge revisions consolidate two or more branches in the version graph into a single trunk. In terms of graph theory, branch revisions are defined by an out-degree greater than 1, while merge revisions have an in-degree greater than 1.

In order to provide helpful information for the user, revisions in the service version graph may be *tagged*. Generally, a revision tag is a string attached to one or more service revisions that describes the revisions functionality, stability, maturity level or any other functional or non-functional aspect.

Figure 2 exemplifies the service version graph for a service named `Service 2`. A number of revision tags (`INITIAL`, `branch_1`, `branch_2`, ...) have been assigned to the graph to identify revisions. The revision tagged `STABLE` is a branch revision; the branches are again merged in revision `LATEST`.

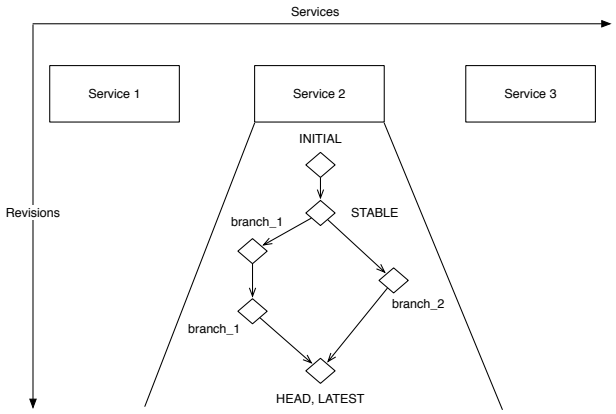

Figure 2. Service Version Graph

2.3. Version Transparency

Optimally, clients in a SOA should be *version-transparent* – version selection, mediation between incompatible versions and automatic rebinding should be handled automatically by the runtime environment. Clients should be able to switch between versions freely, and invoke any revision of the service without adapting the client code.

We envision that this version-transparency can be achieved through *service proxies*. Proxies are bound to a *selection strategy* and update their target revision whenever there is a better match than their current target. Proxies are responsible for mediating between the user-provided data and the expected input of the target revision.

Selection strategies are queries on the service version graph that use the defined revision tags and the inter-version relationships to select the most appropriate service revision for users. A user may, for instance, define a selection strategy that always binds to the most recent revision in the graph, to the latest stable one, or to a revision belonging to a specific branch. Unlike one-time queries, this selection is monitored by the proxy – if the service version graph changes (for instance because of the insertion of a new revision) the result of the selection may also change, and the proxy may update its target service to reflect this change. We refer to this change in the proxies target service as *dynamic rebinding*. Dynamic rebinding is transparent to the client – it can be safely assumed that the most appropriate service according to the selection is invoked.

However, to fully utilize the concept of version transparency it is necessary to turn away from the currently prevalent RPC style of communication. Proxies are, therefore, accessed solely through a simple messaging interface: clients invoke services by passing the service input as an input message to the proxy, and the proxy answers with an output message when the underlying service invocation is

finished. Typically the response message is delivered in a non-blocking fashion. Using selection strategies and the messaging paradigm the tedious details of service versioning can be handled entirely by the proxy.

3. Versioning Support in VRESCo

In the following section we explain how the concepts introduced in Section 2 are implemented within our VRESCo runtime environment.

3.1. VRESCo Runtime

We implemented and evaluated our service versioning approach as part of the VRESCo (Vienna Runtime Environment for Service-Oriented Computing) project. The VRESCo runtime environment has been motivated in [8] and aims at addressing some of the current challenges in Service-oriented Computing research [12] and practice. Among others, this includes topics related to service discovery and metadata, dynamic binding and invocation, service monitoring, Quality of Service (QoS) aware service composition, service management and service notifications. Besides this, another goal of the VRESCo project is to facilitate engineering of service-oriented applications by reconciling some of these topics and abstracting from protocol related issues.

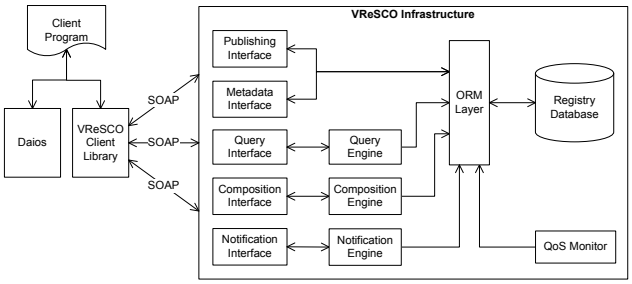

Figure 3. VRESCo Overview

The base architecture of VRESCo is shown in Figure 3. The VRESCo services are provided as Web services which can be accessed either directly using the SOAP protocol, or via the client library that provides a simple API for accessing the VRESCo services.

Services and associated metadata are stored in the registry database that is accessed using the object-relational mapping (ORM) layer. Web services are published and found in the registry using the publishing and querying services. Integrated into the VRESCo system is a QoS monitor as described in [14], which continuously monitors the QoS values of services inside the registry, and keeps the according QoS information in the registry database up to date. Furthermore, the composition engine provides support for composing services using QoS attributes, while the notification engine is responsible for notifying subscribers when certain events of interest occur.

To carry out the actual Web service invocations the DAIOS dynamic Web service invocation framework [7] has been integrated into the VRESCo client-side infrastructure. DAIOS decouples clients from the services they are invoking by abstracting from service implementation issues such as encoding styles, operations or endpoints. Therefore, clients only need to know the address of the WSDL interface describing the target service, and the input message that should be passed to it; all other details of the target service implementation are handled transparently.

3.2. Versioning Metadata

Current service registries such as UDDI do not provide direct support for different versions of the same service. In VRESCo, on the other hand, service versioning is addressed as a first-class concept within the registry. For each service a set of metadata (i.e., data that describes the service's owner, its functionality and purpose) and a service version graph (as defined in Section 2) is stored.

Versioning metadata (i.e., the relationships in the service version graph) are defined by the service provider when publishing new services or revisions. Our implementation does not enforce any specific rules about the degree of change between two revisions in a service version graph. Consequently, it is the provider's decision whether the differences between two service versions are so fundamental that an entire new service should be created instead. However, every evolutionary change as per classification from Section 2.1 mandatorily leads to the creation of either a new revision or a new service. It is not possible to just "update" the interface of an existing service without creating a new revision, even though changes in the implementation of the service are of course possible. However, any evolutionary step may contain multiple discrete changes, i.e., the actual difference between revisions may be arbitrarily complex.

Tag	Description	Assigned by
`INITIAL`	The first version of this service	VRESCo
`STABLE`	A well-tested production-level service version	provider
`HEAD`	The most recent version in a branch	VRESCo
`LATEST`	The most recent version in the entire version graph; implies HEAD	VRESCo
`DEPREC`	The version is online for compatibility reasons, but should not be used anymore (deprecated)	provider
`OFF`	The version has been taken offline and is not available anymore	provider

Table 1. Default revision tags

Establishing branch and merge revisions is also done by the service provider by defining the appropriate relationships in the service version graph. In this regard, our im-

plementation does not impose any further restrictions on branching and merging, i.e., the service providers are free to use branching and merging at their convenience.

VRESCo also supports the concept of revision tags. Tags may either be default tags with a well-defined meaning, or arbitrary strings. A list of default tags currently implemented is given in Table 1. Some of these default tags are assigned automatically by VRESCo (`INITIAL`, `LATEST`, `HEAD`) while others (`STABLE`, `DEPREC`, `OFF`) have to be assigned by the service provider.

3.3. Rebinding Proxies

The VRESCo client library implements the idea of rebinding client proxies as described in Section 2.

Rebinding clients in VRESCo are to a certain degree version-transparent – certain types of version changes can be handled automatically by the runtime environment. That means the same client code may be used to invoke all revisions of the service as long as the difference of the two service revision can be described using only such changes. We refer to these special change types as *transparent change types*. In the classification scheme of Figure 1 we have marked all transparent change types using a light gray background. Change types with dark background are non-transparent – the client code currently has to be adapted for such changes. Types with white background are indefinite – they contain both transparent and non-transparent changes. In a nutshell all change types affecting only optional WSDL parameters, as well as added operations or changed output parameters can be handled transparently, while changes in mandatory parameters are generally non-transparent. Non-functional changes can be handled if the affected non-functional feature is supported by the VRESCo dynamic invoker. Semantic changes are currently out of scope and, therefore, non-transparent. We plan to extend our work on rebinding clients in order to transparently handle all functional change types as part of our future work.

```
1  string domain = "NumberPorting";
2  string selection =
3    "QoS.ResponseTime < 500 and "+
4    "Tag.Name like 'LATEST'";
5  DaiosProxy rebindingClient =
6    ProxyFactory.GetRebindingClient(
7      domain, query, new PeriodicRebinding(5000)
8    );
```

Listing 1. Creating a rebinding client

We give an example of how to create a rebinding proxy within the VRESCo system in Listing 1. The client binds to a service of the domain 'NumberPorting' (within the VRESCo registry services providing the same functionality are grouped into domains) and chooses the most recent version of a service that offers a measured response time of less than 500 ms. Additionally, the client specifies a *rebinding strategy*. The rebinding strategy indicates when the rebinding proxy reconsiders the current binding. In the example the client constructs a periodical rebinding proxy with a rebinding interval of 5000 ms. A list of currently implemented rebinding strategies is given in Table 2.

Strategy	Description
`Fixed`	The proxy never updates its binding
`Periodic`	The proxy reconsiders its binding periodically
`OnDemand`	The proxy reconsiders its binding on client requests
`OnInvocation`	The proxy reconsiders its binding prior to service invocations

Table 2. Rebinding Strategies

All rebinding strategies from Table 2 have their specific advantages and disadvantages. Fixed proxies can be used when rebinding is not desired in a specific scenario (e.g., because of existing contractual obligations). Periodic rebinding causes constant overhead, but is inefficient if invocations happen only sparsely whereas on demand rebinding causes very low overhead but is not always accurate. Finally, on invocation rebinding is guaranteed to be accurate but may seriously degrade the invocation time.

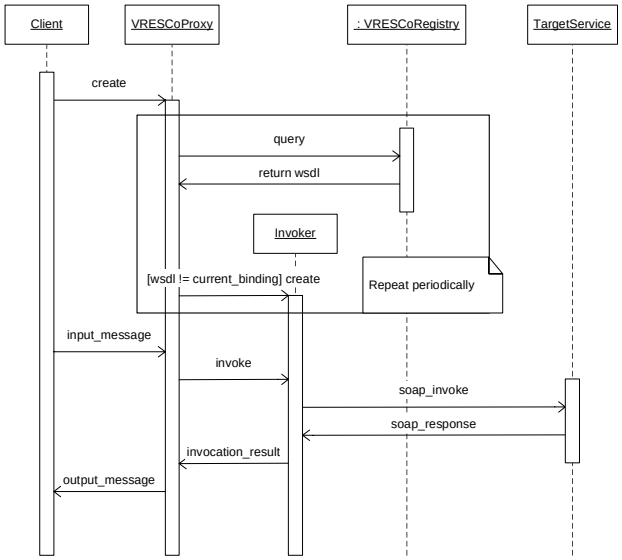

Figure 4. Periodic Rebinding Proxy

Figure 4 exemplifies the internal processing of a periodical rebinding proxy: the proxy is initialized with a domain, a selection strategy and a rebinding interval. The proxy then periodically queries the VRESCo registry according to the rebinding interval, and checks if the current binding is still accurate. If there is a new service which better matches according to the selection strategy, the proxy discards the current binding (i.e., destroys the current `Invoker` object)

and constructs a new `Invoker` pointing to the new service. If the client finally initiates a service invocation by passing an input message to the proxy it forwards the request to the invoker, which dynamically constructs an invocation fitting to the service using the client input data, fires the invocation, receives the result and asynchronously passes it back to the client.

Mediation between the client message and service interface happens in the dynamic invocation phase and is handled by the invoker. More details on this dynamic invocation mechanism can be found in [7].

4. Evaluation

To demonstrate our approach to Web service versioning, we use a case study from the telecommunications domain which is derived from [8].

4.1. Case Study

In this paper we consider a telecommunications provider (TELCO) that provides a telephone number porting Web service to its competitors. In its beginning, the service was implemented using the state of the art technology of that time (i.e., JAX-RPC using Apache Axis[2]). In these days, the interface was intentionally kept basic, including only the number to port and the new provider as input, and a confirmation of the successful porting operation as output.

However, not all potential users of the service (i.e., other providers) were satisfied with the simple service interface: some providers' business processes intended to send additional customer information (e.g., name, address), and in some special occasions, an indirect port via a third party was necessary. After some discussion, a new variant of the service was created with an extended interface that included this additional information.

Even worse, after a company merge the new IT management of the TELCO decided to switch to Microsoft's .NET platform. Both service flavors were, therefore, ported to .NET and the Windows Communication Foundation (WCF) platform[3], a step that was not received benevolently by all users of the service. It was, therefore, agreed to keep the original JAX-RPC services online for some time.

Both variants of the service had to be adapted one more time – the initial definition of the confirmation turned out to be too limited, and had to be expanded. A new field containing additional textual description was, therefore, added.

Eventually, a last evolutionary step was necessary: number porting was getting more and more popular over time, and the server hosting the number porting partner services could not deal with the load; the original server was therefore replaced by a more powerful service host. However, the old server was kept online as a fallback solution for the not yet fully tested new machine.

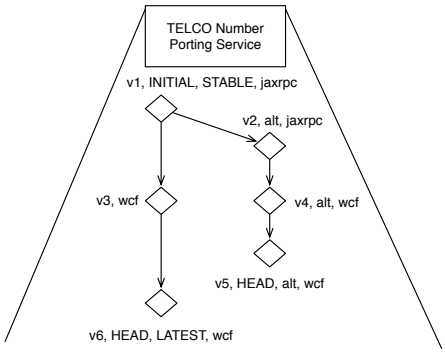

Figure 5. Scenario implementation

Figure 5 illustrates this case study using the notions introduced in Section 2. The service version graph in this figure contains the six distinct service revisions, which are arranged in two different branches. A number of revision tags have been added to the various revisions: `jaxrpc` or `wcf` to describe the technological platform, `alt` to identify services of the alternate branch, and `HEAD` to specify the most recent versions in both branches. The first version is tagged `STABLE`. The tags `v1` to `v6` have been introduced to serve as human-readable version identifiers.

In Table 3 the changes from our case study are categorized according to the classification from Section 2. It is important to note that all these changes are transparent.

Base Revision	New Revision	Change Type
v1	v2	Added optional parameters
v1	v3	Non-Functional Change (platform change)
v2	v4	Non-Functional Change (platform change)
v4	v5	Changed output type
v3	v6	Non-Functional Change (server relocation)

Table 3. Version differences

In order to evaluate our service versioning scheme and to demonstrate its advantages, we apply this case study to compare our approach to established state of the art technology: we use UDDI to store and manage the different versions of the number porting Web service, and the Apache Web Service Invocation Framework WSIF[4] to carry out Web service invocations.

4.2. Registry Versioning Support

Supporting different versions of the same service is no built-in feature of UDDI. Storing a version graph such as

[2] http://ws.apache.org/axis/
[3] http://wcf.netfx3.com/
[4] http://ws.apache.org/wsif/

the one discussed in our case study, therefore, demands for suitable and agreed conventions on how to represent the the graph in the flat UDDI registry structure (see for instance [2]).

Using such a solution versioning is actually implemented on client-side. From a UDDI registry perspective, all services are completely independent; it is the client who decides based on conventions which revisions belong to the same base service, and how the revisions are interrelated. Given that service versioning clearly is a registry issue this is no good separation of concerns, and complicates the implementation of the client. Another evident disadvantage of this solution is that all clients of the UDDI registry need to have the same understanding of the representation conventions used in the registry.

With VRESCo none of these problems arise since versioning is explicitly handled by the SOA runtime. VRESCo clients to not need to care about versioning issues or conventions, but can rely on the SOA runtime to bind them to the most appropriate service revision, based on their selection strategy.

4.3. Dynamic Invocation

Implementing a dynamic Web service client that is able to invoke services discovered at run-time (e.g., by querying a registry) is not easy with current technologies such as Apache WSIF [7]. Even though WSIF provides a dynamic (stubless) interface it still needs service- and revision-specific information: the WSDL description address and the functional interface of the service. The WSDL address can be retrieved directly from the registry, but the functional interface needs to be parsed from the WSDL description manually prior to constructing the actual invocation which is rather cumbersome and error-prone.

What is even more problematic is WSIF's often discussed inability to dynamically invoke services that take complex XML types as parameter or return such. In our case study we use complex types to represent service inputs and outputs, and so do most real-life business Web services. Consequently, WSIF clients are usually hard-wired to a certain pre-selected service revision using compiled stubs. If the client application has to be migrated to a different revision a re-engineering of the client application is necessary. Switching service revisions (for instance upgrading to a newer version) is not possible at run-time.

VRESCo, on the other hand, needs only the WSDL address to construct a dynamic frontend to any revision of the service. Since all changes in our case study are transparent it is possible to use the same client code to invoke any revision of the number porting service. Migrating the application to a different version or branch in the service version graph is only a question of selecting and binding to a different revision, the actual invocation does not need to be changed at all. Furthermore, XML complex types do not pose a problem for our implementation.

4.4. Dynamic Rebinding

One of the core features of the VRESCo versioning support is its ability to decouple service clients from the concrete service version they are using by means of version-transparent proxies. Therefore, in our approach clients can switch between different versions at runtime, without adaptation of the client code. Neither Apache WSIF nor UDDI support similar concepts. In order to simulate such functionality using these technologies the client application would have to handle all low-level issues of service versioning and dynamic rebinding (i.e., permanent polling of the registry, the actual rebinding process) itself.

5. Related Work

The evolution of Web services is subject to a wide ranging debate. The World Wide Web Consortium recently paid attention to versioning of Web services [16, 17]. Currently, a common workaround to deal with the lack of versioning support is to use separate namespaces for each version of a Web service.

The general problem of versioning of distributed software systems is sketched in [15] where the author distinguishes between *Distributed-Object Versioning* and *Messaging Versioning*. Even more generally, Web service evolution can be considered a special case of the Software Configuration Management (SCM) problem, which has already been extensively researched [4]. Therefore, we have adopted many notions from SCM (e.g., revisions, branching and merging, revision tagging) in our service evolution approach.

One approach that addresses Web service evolution has been introduced by Kaminski et al. [6]. The authors outline various requirements for versioning, and demonstrate why common versioning strategies are inappropriate in the context of Web services. Instead they propose to use the *Chain of Adapters* pattern [5] for developing evolving Web services.

Ponnekanti et al. [13] address the interoperability among independently evolving Web services The authors introduce static and dynamic analysis algorithms to identify compatibility between applications and non-native services, and present tools that implement these algorithms. Moreover, they introduce so called "cross stubs" for resolving incompatibilities.

Current registry standards provide only little support for evolving Web services. The ebXML versioning approach [10] is based upon the versioning extensions of Web-

DAV [3], but provides only a small subset of this functionality. If the ebXML registry supports versioning, all registry items are implicitly under version-control. However, it should be noted that ebXML mainly focuses on versioning of registry data but it remains unclear how clients can access specific service revisions.

In contrast to ebXML, the UDDI specification [11] does not mention service versioning at all. One common approach to use versioning in UDDI is based on the prerequisite that a given version of a `wsdl:portType` should be represented by a unique `tModel`. Current best practices for service versioning using UDDI are described in [2].

6. Conclusion

Factors such as new requirements or platforms, or the need to correct faults in previous releases, cause software systems to evolve over time, and new versions of systems to be released. In service-oriented computing, versioning represents an even more important issue since changes in services usually directly affect service requesters. In this paper, we presented a general approach to versioning of service-oriented systems using service version graphs and selection strategies, and demonstrated how this approach is implemented in our VRESCo service runtime environment. Our approach leverages the dynamic invocation mechanism of VRESCo that enables to invoke different versions using transparently rebinding service proxies. We illustrated the advantages of our approach in comparison to UDDI and Apache WSIF using a case study from the telecommunications domain.

As part of our future work we plan to further extend the concept of dynamic rebinding – currently only a subset of all possible interface change types can be handled transparently. Other changes still require adaption of the service client code, which is often undesired. We therefore want to enhance the VRESCo versioning model to support not only inter-version relationships, but also the actual version deltas (i.e., differences between revisions), in order to be able to implement full run-time mediation for non-transparent changes.

References

[1] K. H. Bennett and V. T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM.

[2] K. Brown and M. Ellis. Best Practices for Web Services Versioning, 2004. `http://www-128.ibm.com/developerworks/webservices/library/ws-version/` (Last accessed: January 14, 2008).

[3] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning), 2002. `http://www.webdav.org/deltav/protocol/rfc3253.html` (Last accessed: Jan. 14, 2008).

[4] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[6] P. Kaminski, H. Müller, and M. Litoiu. A Design for Adaptive Web Service Evolution. In *Proceedings of the International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS'06)*, pages 86–92, New York, NY, USA, 2006. ACM Press.

[7] P. Leitner, F. Rosenberg, and S. Dustdar. Daios – Efficient Dynamic Web Service Invocation. Technical Report TUV-1841-2007-01, Vienna University of Technology, 2007.

[8] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar. Towards Recovering the Broken SOA Triangle – A Software Engineering Perspective. In *Proceedings of the Second International Workshop on Service Oriented Software Engineering (IW-SOSWE'07)*, pages 22–28, Sept. 2007.

[9] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Adaption for WS-BPEL. In *Proceedings of the 17th International World Wide Web Conference (WWW'08), Beijing, China*, Apr. 2008. to appear.

[10] OASIS International Standards Consortium. *ebXML Registry Services and Protocols v3.0*, Mar. 2005.

[11] OASIS International Standards Consortium. *Universal Description, Discovery and Integration v3.0 (UDDI)*, Feb. 2005.

[12] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 11, 2007.

[13] S. R. Ponnekanti and A. Fox. Interoperability Among Independently Evolving Web Services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware (Middleware'04)*, pages 331–351, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[14] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, Sept. 2006.

[15] S. Vinoski. The More Things Change .... *IEEE Internet Computing*, 8(1):87–89, 2004.

[16] World Wide Web Consortium. *Extending and Versioning Languages*, July 2006. `http://www.w3.org/2001/tag/doc/versioning-20060726.html` (Last accessed: January 14, 2008).

[17] World Wide Web Consortium. *Web Service Description Language (WSDL) Version 2.0 Primer*, Mar. 2006. `http://www.w3.org/TR/wsdl20-primer/` (Last accessed: January 14, 2008).

[18] World Wide Web Consortium (W3C). *Semantic Annotations for WSDL and XML Schema*, 2007. `http://www.w3.org/TR/sawsdl/` (Last accessed: Jan. 23, 2008).

[19] World Wide Web Consortium (W3C). *Simple Object Access Protocol v1.2 (SOAP)*, 2007. `http://www.w3.org/TR/soap/` (Last accessed: Jan. 23, 2008).